Model Transformations

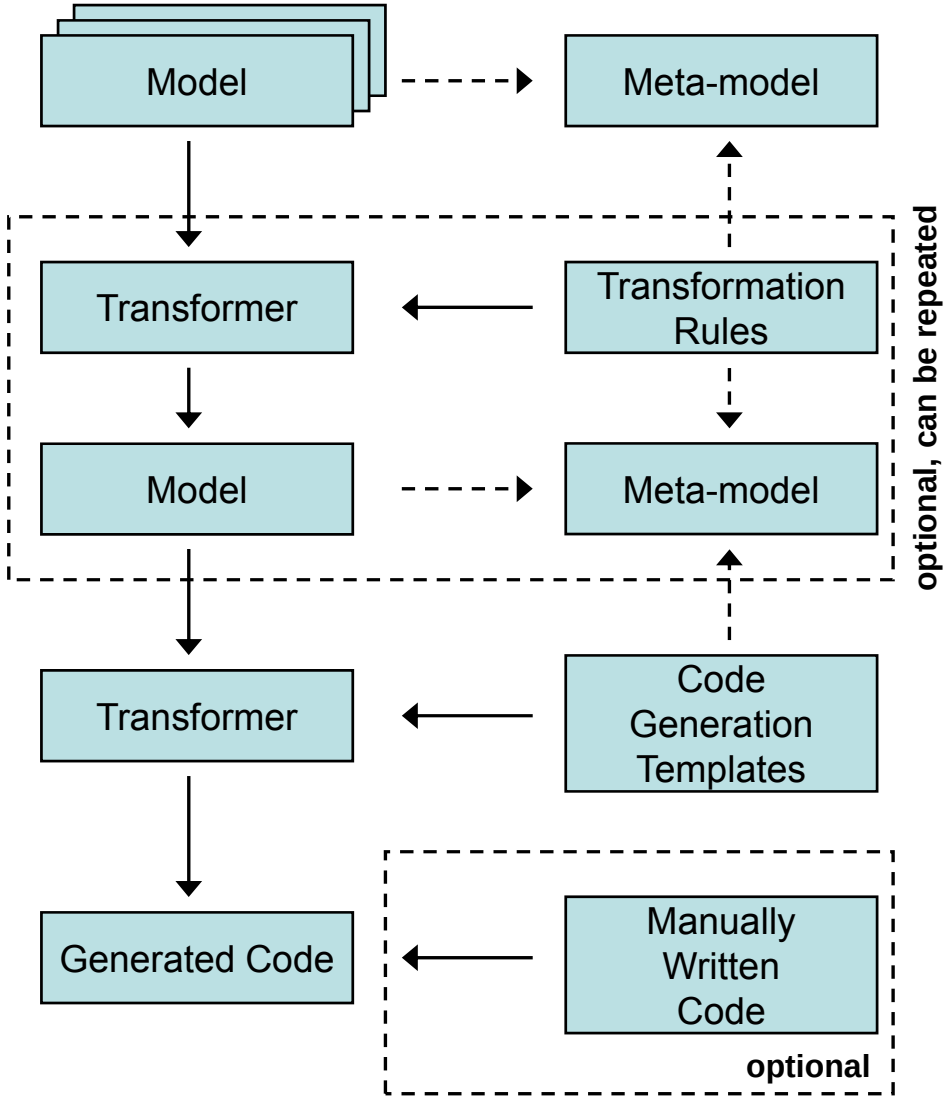
What is a transformation?

- A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition.
- A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.
- A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.
  - Unambiguous specifications of the way that (part of) one model can be used to create (part of) another model

- Preferred characteristics of transformations
  - **semantics-preserving**

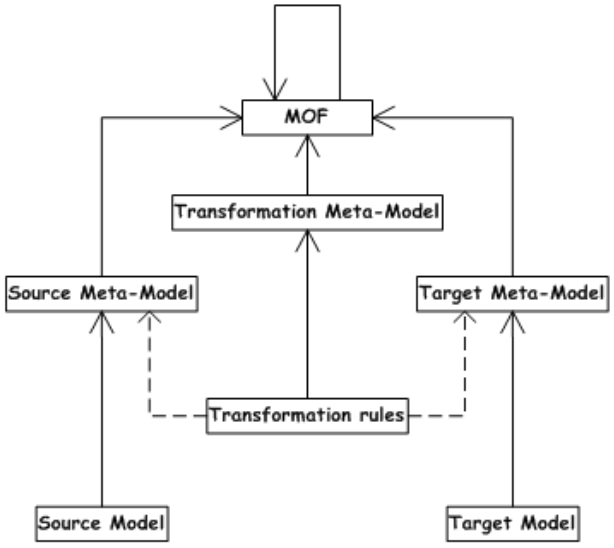
Model-to-model vs. Model-to-code

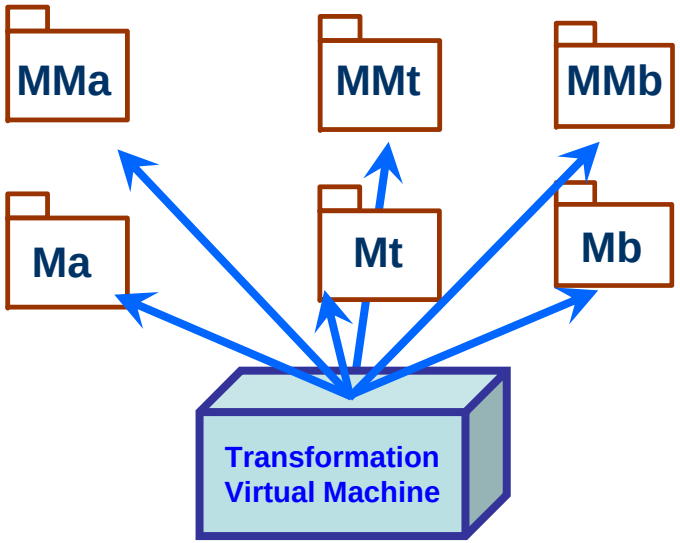
- **Model-to-model** transformations
  - Transformations may be between different languages. In particular, between different languages defined by MOF
- **Model-to-text** transformations
  - Special kind of model to model transformations
  - MDA TS to Grammar TS



Transformations as models

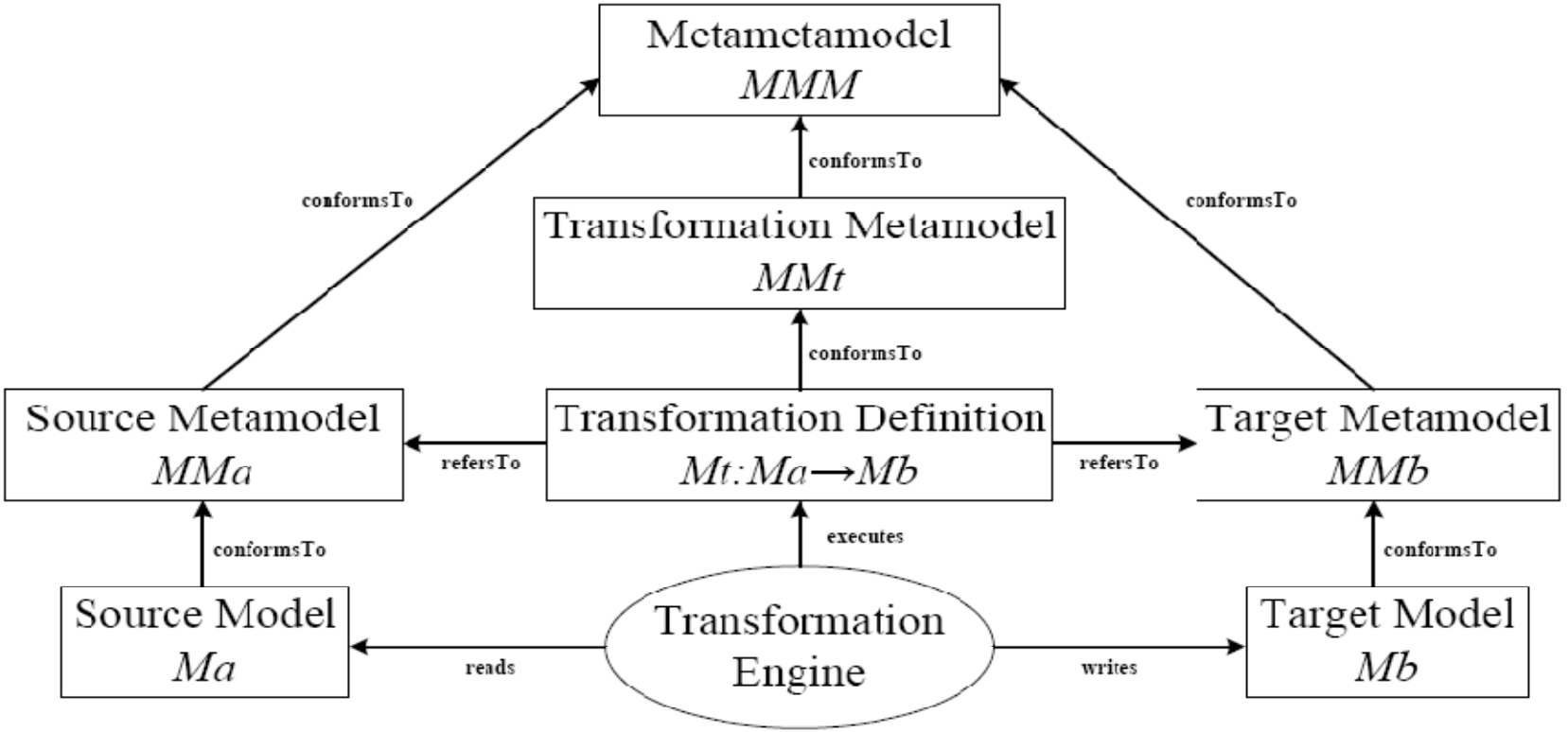
- Treating everything as a model leads not only to conceptual simplicity and regular architecture, but also to implementation efficiency.
- An implementation of a transformation language can be composed of a transformation virtual machine plus a metamodel-driven compiler.
- The transformation VM allows uniform access to model and metamodel elements.





Model transformation

- Each model conforms to a metamodel.
- A transformation builds a target model (Mb) from a source model (Ma).
- A transformation is a model (Mt, here) conforming to a metamodel (MMt).



Characterisation of model transformations (1)

- **Endogenous** vs. **exogenous**
  - **Endogenous** transformations are transformations between models expressed in the same metamodel. Endogenous transformations are also called **rephrasing**
    - Optimisation, refactoring, simplification, and normalization of models.
  - Transformations between models expressed using different meta-models are referred to as **exogenous** transformations or **translations**
    - Synthesis of a higher-level specification into a lower-level one, reverse engineering, and migration from a program written in one language to another

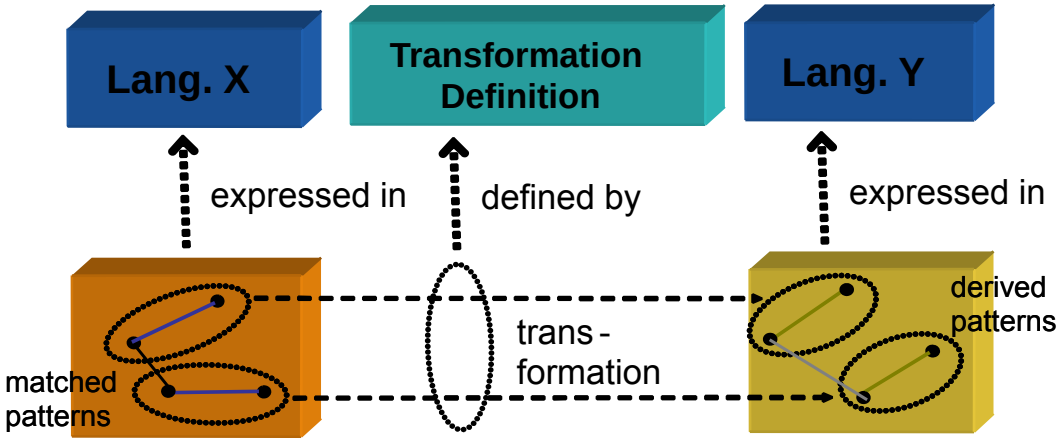
- **Horizontal** vs. **vertical**
  - **Horizontal** transformations are transformations where the source and target models reside at the same abstraction level
    - Refactoring (an endogenous transformation) and migration (an exogenous transformation)
  - **Vertical** transformations are transformation where the source and target models reside at different abstraction levels
    - Rrefinement, where a specification is gradually refined into a full-fledged implementation

Characterisation of model transformations (2)

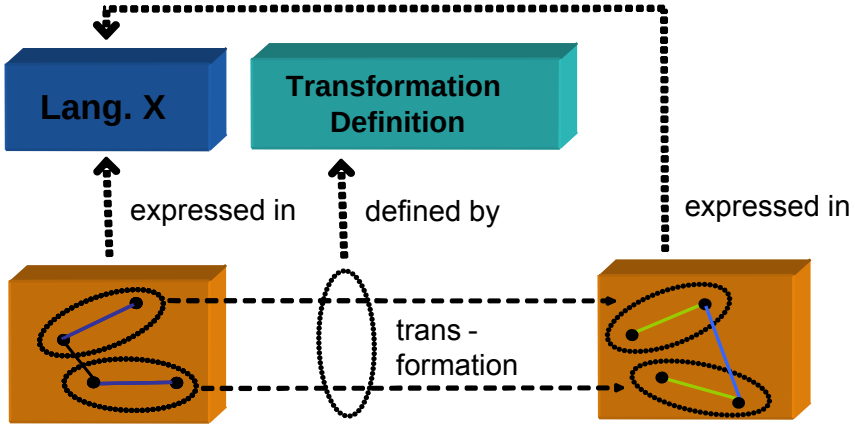
- **Level of automation**
  - The level of automation is the grade to which a model transformation can be automated.
- **Complexity**
  - Simple transformations
    - Mappings for identifying relations between source and target model elements
  - Complex transformations
    - Synthesis, where higher-level models are refined to lower-level models
- **Preservation**
  - Each transformation preserves certain aspects of the source model in the transformed target model.
  - The properties that are preserved can differ significantly depending on the type of transformation.
    - With refactorings the (external) behaviour needs to be preserved, while the structure is modified.
    - With refinements, the program correctness needs to be preserved.

Characterisation of model transformations (3)

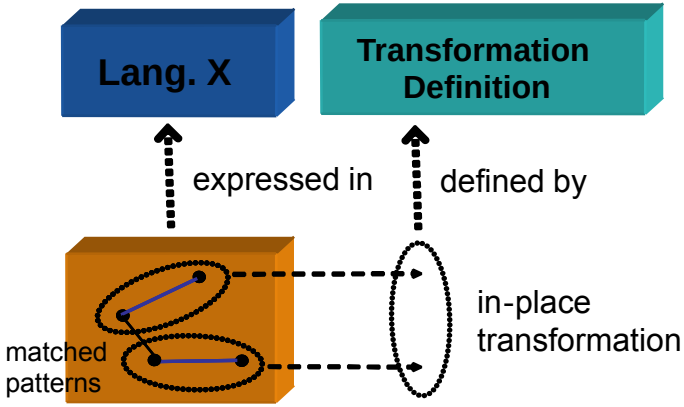
Transformation = Matching and deriving patterns



Transformation in the same meta-model

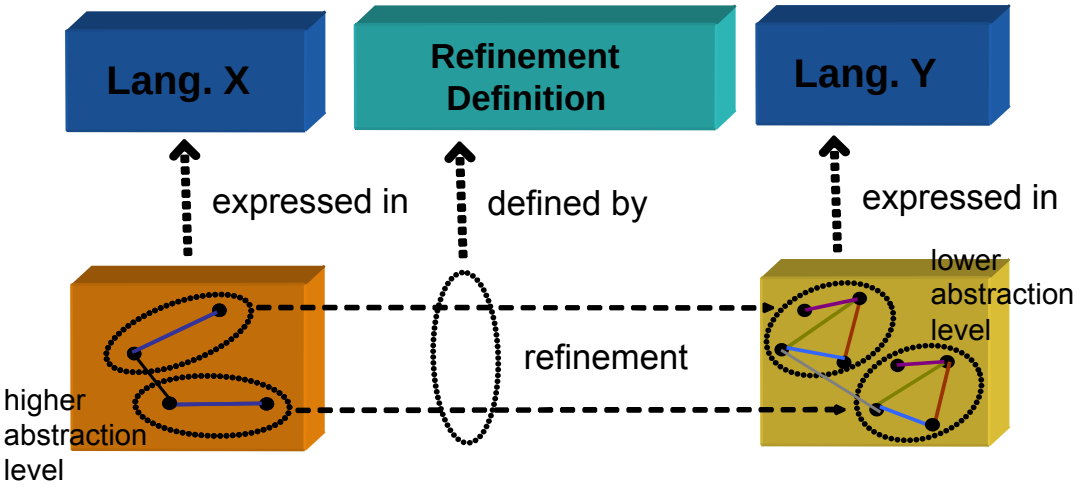


Transformation in the same model

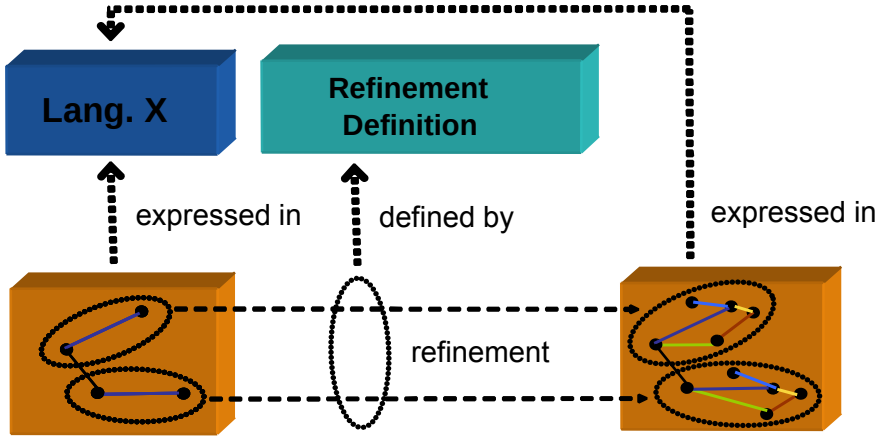


Characterisation of model transformations (4)

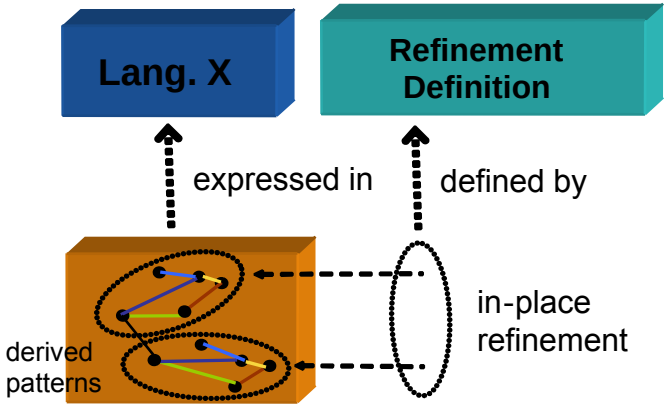
Refinement preserve meaning and derives complex patterns



Refinement in the same meta-model

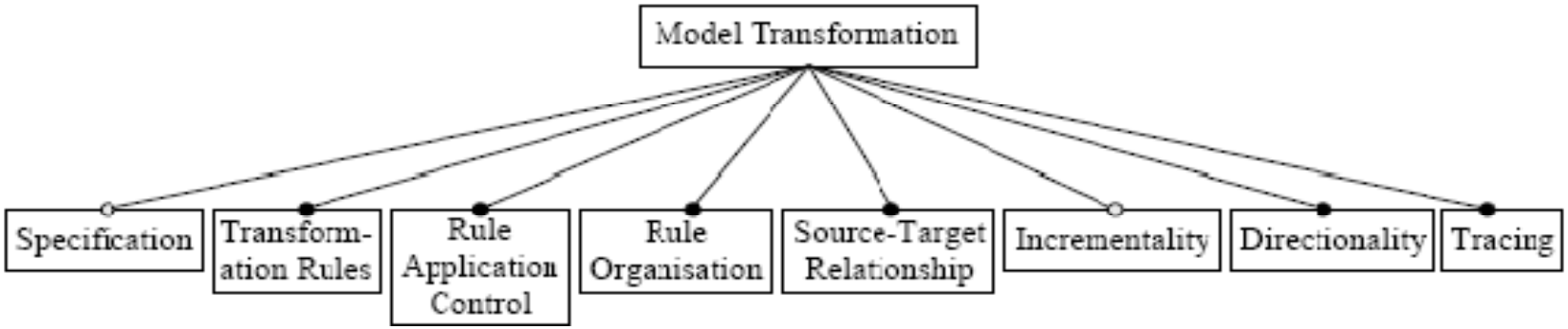


Refinement in the same model



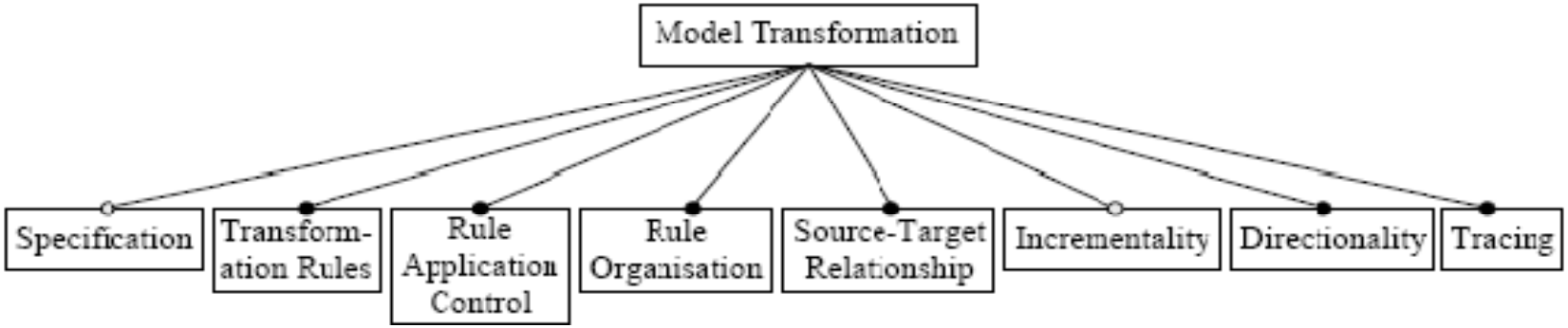
Features of model transformations

- **Specification**
  - Some approaches provide a dedicated specification mechanism, such as pre-/post-conditions expressed in OCL.
- **Transformation rules**
  - A transformation rule consists of two parts:
    - A left-hand side (LHS), which accesses the source model
    - A right-hand side right-hand side (RHS), which expands in the target model
  - A **domain** is the rule part used for accessing the models on which the rule operates
  - The **body** of a domain can be divided into three subcategories
    - Variables: Variables may hold elements from the source and/or target models
    - Patterns: Patterns are model fragments with zero or more variables
    - Logic:. Logic expresses computations and constraints on model elements
  - The transformations variables and patterns can be **typed**.



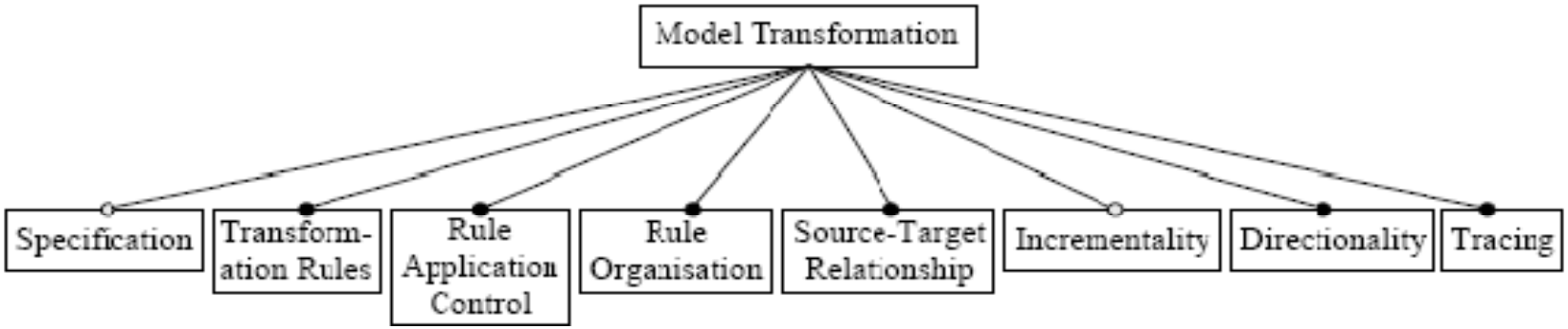
Features of model transformations

- **Rule application control**
  - **Location determination** is the strategy for determining the model locations to which transformation rules are applied.
  - **Scheduling** determines the order in which transformation rules are executed.
- **Rule organisation**
  - Rule organisation is concerned with composing and structuring multiple transformation rules by mechanisms such as modularisation and reuse.
- **Source-target relationship**
  - whether source and target are one and the same model or two different models
    - Create new models
    - Update existing models
    - In-place update



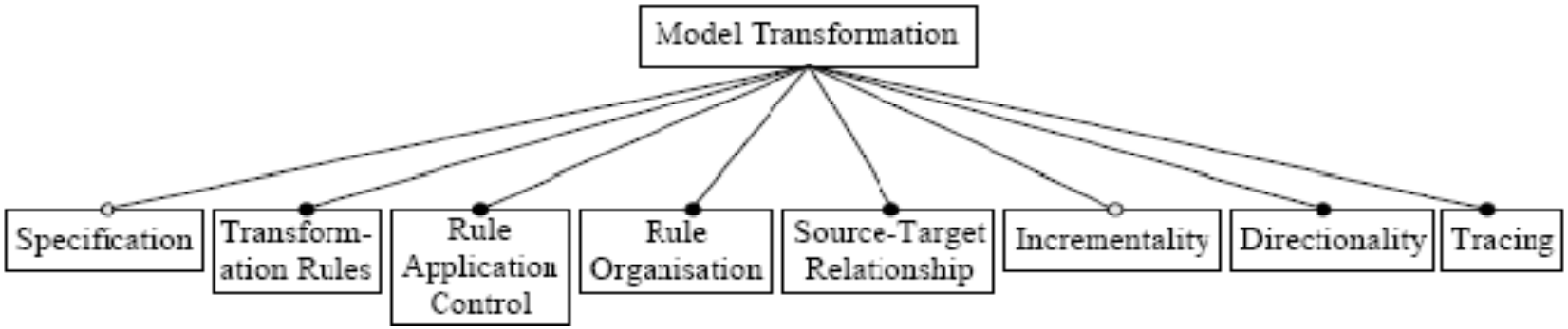
Features of model transformations

- **Incrementality**
  - Ability to update existing target models based on changes in the source models
- **Directionality**
  - Unidirectional transformations can be executed in one direction only, in which case a target model is computed (or updated) based on a source model
  - Multidirectional transformations can be executed in multiple directions, which is particularly useful in the context of model synchronisation.



Features of model transformations

- **Tracing**
  - Mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements.
  - Trace information can be useful in
    - performing impact analysis (i.e. analyzing how changing one model would affect other related models),
    - determining the target of a transformation as in model synchronization
    - model-based debugging (i.e. mapping the stepwise execution of an implementation back to its high-level model)
    - debugging model transformations themselves



Model-to-model approaches (1)

- **Direct manipulation approaches**
  - Offers an internal model representation and some APIs to manipulate it
  - Usually implemented as an object-oriented framework
  - Users usually have to implement transformation rules, scheduling, tracing, etc.
  - Examples: Java Metadata Interface (JMI), EMF, …

- **Structure-driven approaches**
  - Two distinct phases:
    - The first phase is concerned with creating the hierarchical structure of the target model
    - The second phase sets the attributes and references in the target
  - The overall framework determines the scheduling and application strategy; users are only concerned with providing the transformation rules
  - Example: OptimalJ

Model-to-model approaches (2)

- **Template-based approaches**
  - Model templates are models with embedded meta-code that compute the variable parts of the resulting template instances.
  - Model templates are usually expressed in the concrete syntax of the target language, which helps the developer to predict the result of template instantiation
  - Typical annotations are conditions, iterations, and expressions, all being part of the meta-language. An expression language to be used in the meta-language is OCL.
  - Examples: Czarnecki, Antkiewicz (2005)

- **Operational approaches**
  - Similar to direct manipulation but offer more dedicated support for model transformation
  - Extend the utilized metamodeling formalism with facilities for expressing computations
    - Extend a query language such as OCL with imperative constructs.
    - The combination of MOF with such extended executable OCL becomes a fully-fledged object-oriented programming system.)
  - Examples: QVT Operational mappings, XMF-Mosaic's executable MOF, MTL, C-SAW, Kermeta, etc.

Model-to-model approaches (3)

- **Relational approaches**
  - Declarative approaches in which the main concept is mathematical relations
  - The basic idea is to specify the relations among source and target element types using constraints
  - Since declarative constraints are non-executable, declarative approaches give them an executable semantics, such as in logic programming
  - Relational approaches are side-effect-free, support multidirectional rules, can provide backtracking …
  - Examples: QVT Relations, MTF, Kent Model Transformation Language, Tefkat, AMW, mappings in XMF-Mosaic, etc.

Model-to-model approaches (4)

- **Graph-transformation-based approaches**
  - Based on the theoretical work on graph transformations
  - Operates on typed, attributed, labelled graphs
  - Graph transformation rules have an LHS and an RHS graph pattern.
    - The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place
    - Additional logic, for example, in string and numeric domains, is needed to compute target attribute values such as element names
  - Examples: AGG, AToM3, VIATRA, GReAT, UMLX, BOTL, MOLA, Fujaba, etc.

Model-to-model approaches (5)

- **Hybrid approaches**
  - Hybrid approaches combine different techniques from the previous categories
    - as separate components
    - or/and , in a more fine-grained fashion, at the level of individual rules
  - In a hybrid rule, the source or target patterns are complemented with a block of imperative logic which is run after the application of the target pattern
  - Rules are unidirectional and support rule inheritance.
  - Examples:
    - Separate components: QVT (Relations, Operational mappings, and Core)
    - Fine-grained combination: ATL and YATL

Model-to-model approaches (6)

- **Other approaches**
  - Extensible Stylesheet Language Transformation (XSLT)
    - Models can be serialized as XML using the XMI
    - Model transformations can be implemented with Extensible Stylesheet Language Transformation (XSLT), which is a standard technology for transforming XML
    - The use of XMI and XSLT has scalability limitations
    - Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations
  - Application of meta-programming to model transformation
    - Domain-specific language for model transformations embedded in a meta-programming language.

Model-to-text approaches

- **Visitor-based** approaches
  - Use visitor mechanism to traverse the internal representation of a model and write text to a text stream
  - Example: Jamda

- **Template-based** approaches
  - The majority of currently available MDA tools support template-based model-to-text generation
    - structure of a template resembles more closely the code to be generated
    - Templates lend themselves to iterative development (they can be derived from examples)
  - A template consists of the target text containing slices of meta-code to access information from the source
  - Examples: oAW, JET, Codagen Architect, AndroMDA, ArcStyler, MetaEdit, OptimalJ, etc.

QVT Operational

MOF QVT: OMG's model-to-model transformation standard

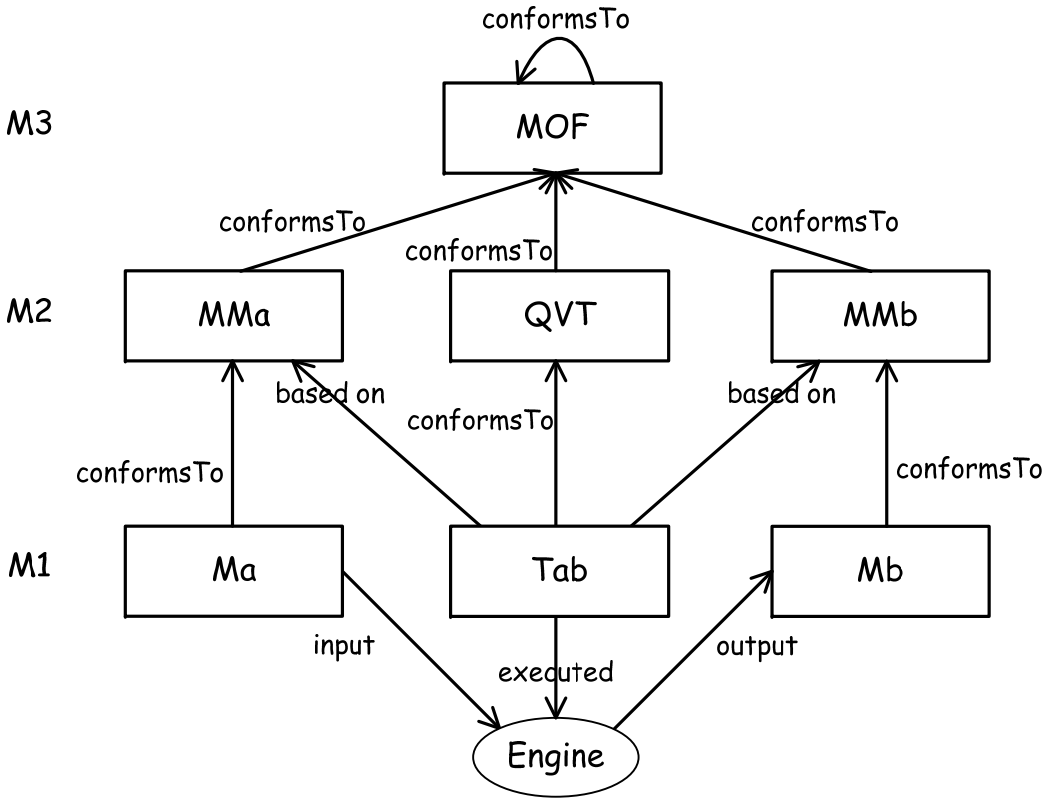
- **QVT** stands for **Q**uery/**V**iews/**T**ransformations
  - OMG standard language for expressing *queries*, *views*, and *transformations* on MOF models

- OMG QVT Request for Proposals (QVT RFP, ad/02-04-10) issued in 2002
  - Seven initial submissions that converged to a common proposal
  - Current status (June, 2011): version 1.1, formal/11-01-01

http://www.omg.org/spec/QVT/1.0/
http://www.omg.org/spec/QVT/1.1/

MOF QVT context

- Abstract syntax of the language is defined as MOF 2.0 metamodel
  - Transformations (*Tab*) are defined on the base of MOF 2.0 metamodels (*MMa*, *MMb*)
  - Transformations are executed on instances of MOF 2.0 metamodels (*Ma*)



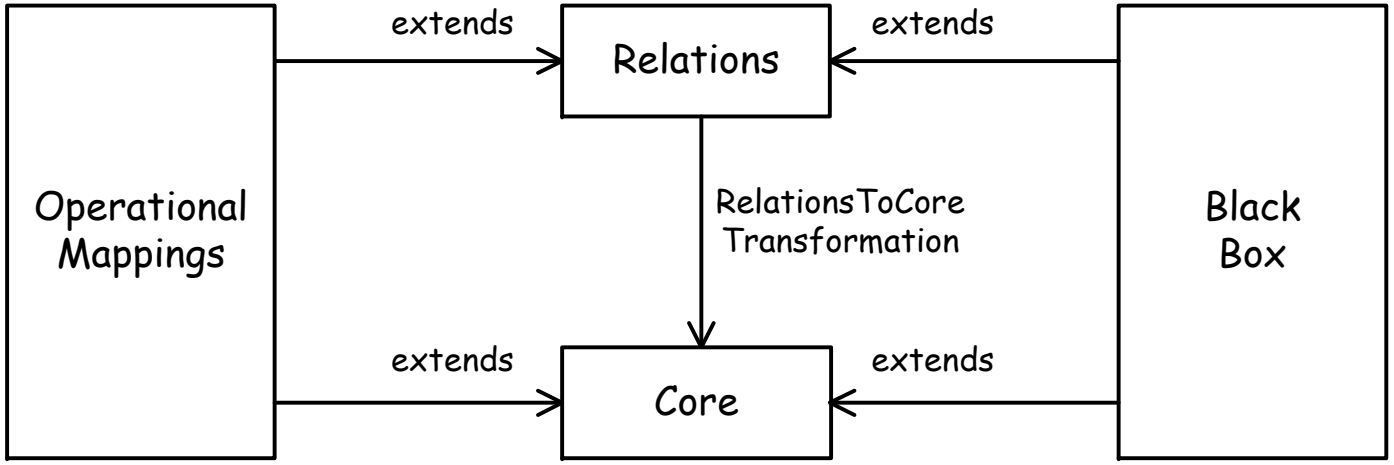
Requirements for MOF QVT language

- Some requirements formulated in the QVT RFP

Mandatory requirements	
Query language	Proposals shall define a language for querying models
Transformation language	Proposals shall define a language for transformation definitions
Abstract syntax	The abstract syntax of the QVT languages shall be described as MOF 2.0 metamodel
Paradigm	The transformation definition language shall be declarative
Input and output	All the mechanisms defined by proposals shall operate on models instances of MOF 2.0 metamodels
Optional requirements	
Directionality	Proposals may support transformation definitions that can be executed in two directions
Traceability	Proposals may support traceability between source and target model elements
Reusability	Proposals may support mechanisms for reuse of transformation definitions
Model update	Proposals may support execution of transformations that update an existing model

MOF QVT architecture

- Layered architecture with three transformation languages:
  - **Relations** (declarative)
  - Core (declarative, simpler than Relations)
  - **Operational Mappings** (imperative)
- Black Box is a mechanism for calling external programs during transformation execution
- QVT is a set of three languages that collectively provide a hybrid “language”.



Overview of Operational Mappings (OM)

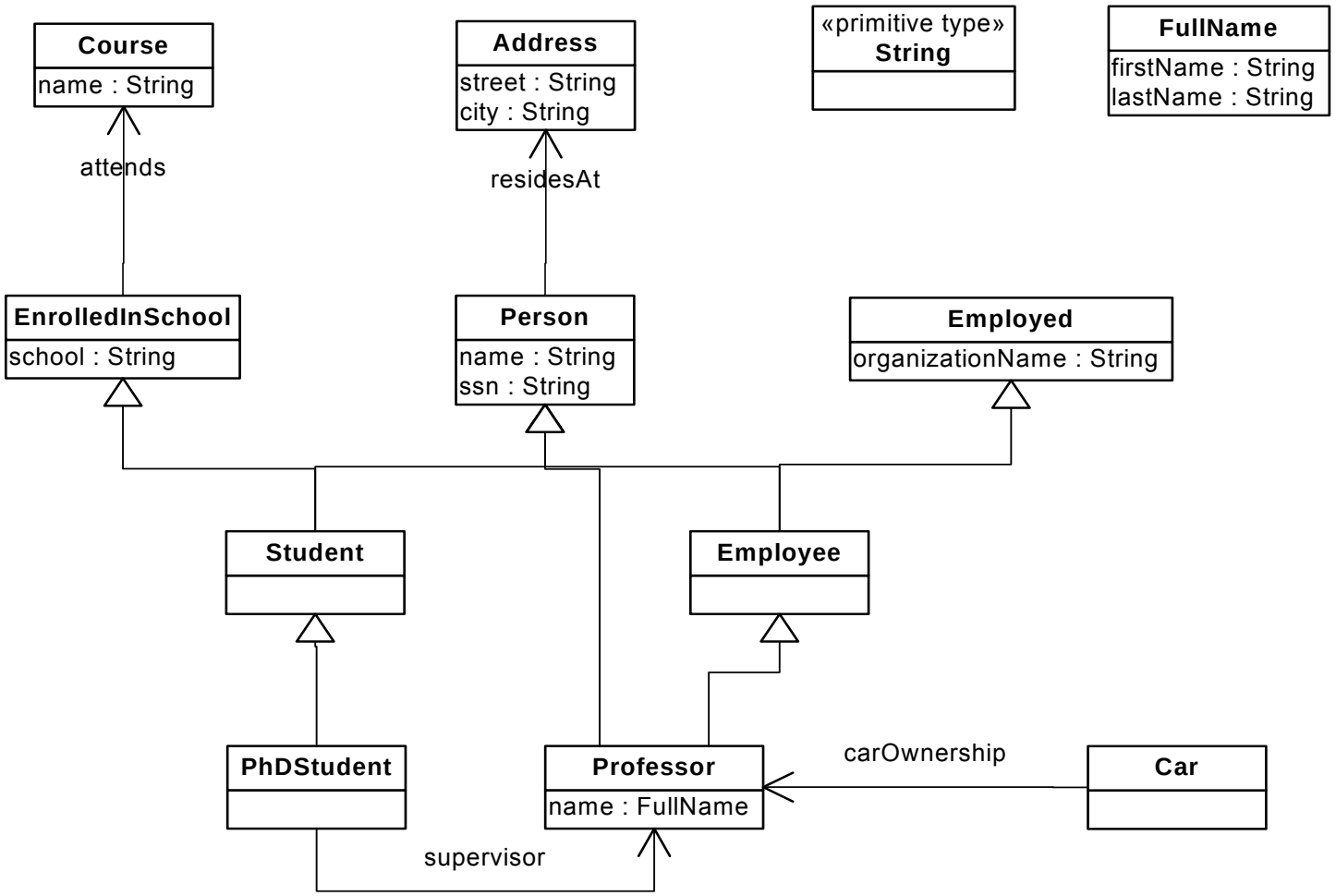
- Imperative transformation language that extends relations
- OM execution overview:
  - **Init**: code to be executed before the instantiation of the declared outputs.
  - **Instantiation** (internal): creates all output parameters that have a null value at the end of the initialization section
  - **Population**: code to populate the result parameters and the
  - **End**: code to be executed before exiting the operation. Automatic handling of traceability links
- Transformations are unidirectional
- Supported execution scenarios:
  - Model transformations
  - In-place update
- OM uses explicit internal scheduling, where the sequence of applying the transformation rules is specified within the transformation rules
- Updates have to be implemented in the model transformations

Flattening class hierarchies example

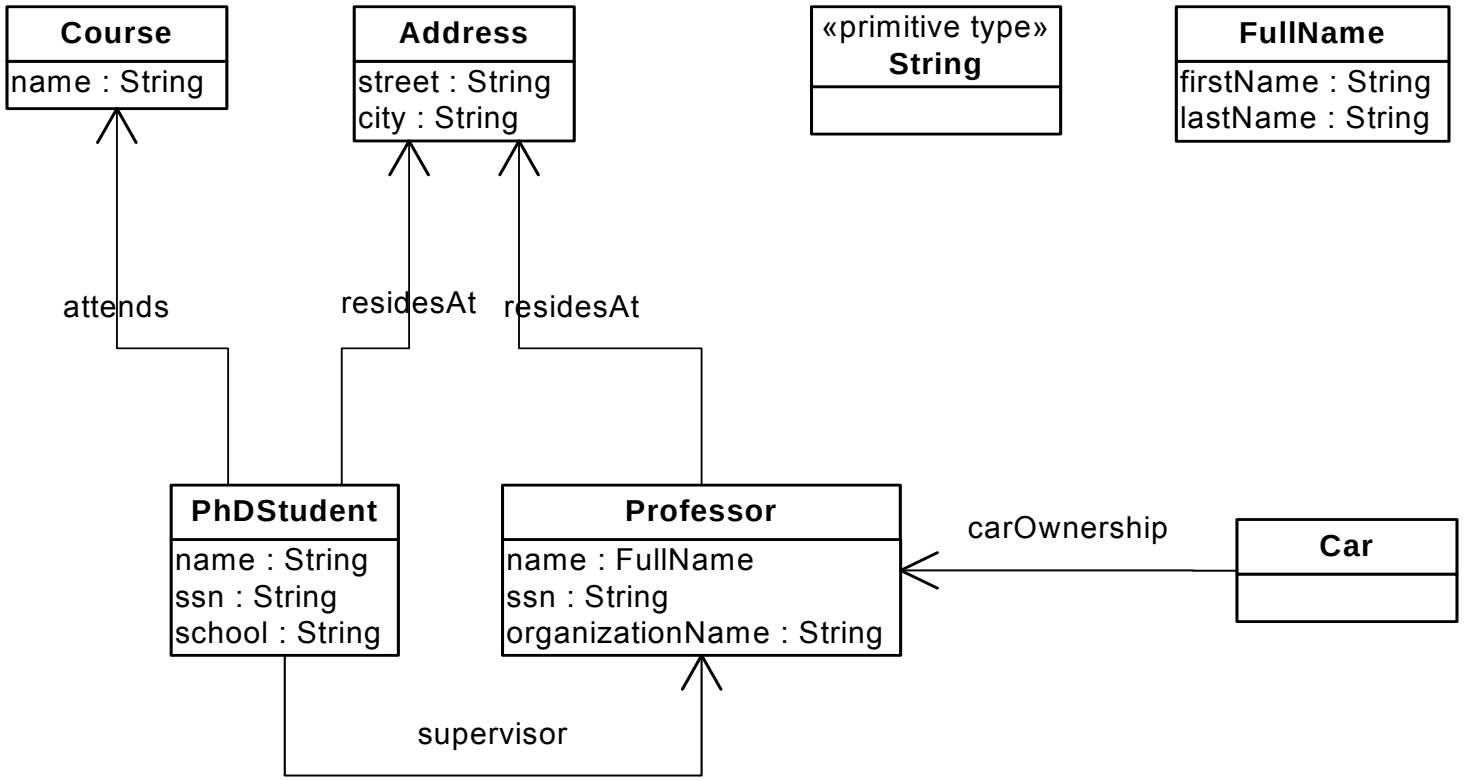
- Flattening UML class hierarchies: given a source UML model transform it to another UML model in which only the leaf classes (classes not extended by other classes) in inheritance hierarchies are kept.

- Rules:
  - Transform only the leaf classes in the source model
  - Include the inherited attributes and associations
  - Attributes with the same name override the inherited attributes
  - Copy the primitive types

Sample input model



Sample output model



OM language: Transformation program structure

```
transformation
flatten
(in hierarchical : UML,
 out flat : UML);
```

Signature: declares the transformation name and the source and target metamodels. **in** and **out** keywords indicate source and target model variables.

```
main() {
  …
}
…
```

Entry point: execution of the transformation starts here by executing the operations in the body of `main`

helper declarations

…

mapping operations declarations

Transformation elements: Transformation consists of mapping operations and helpers forming the transformation logic.

Mapping operations

- A mapping operation maps one or more source elements into one or more target elements
- Always unidirectional
- Selects source elements on the base of a type and a Boolean condition (guard)
- Executes operations in its body to create target elements
- May invoke other mapping operations and may be invoked
- Mapping operations may be related by inheritance, merging, and disjunction

General structure of mapping operations

```
mapping Type::operationName(((in|out|inout) pName : pType)*)
  : (rName : rType)+
  when {guardExpression}                pre-condition
  where {guardExpression} {             post-condition
  init {
    …                                   init section contains code executed before the instantiation of the declared result
  }                                     elements
```

There exists an implicit instantiation section that creates all the output parameters not created in the init section. The trace links are created in the instantiation section.

```
  population {                          population section contains code that sets the values or the result and the
    …                                   parameters declared as out or inout. The population keyword may be
  }                                     skipped. The population section is the default section in the operation body.

  end {
  …                                     end section contains code executed before exiting the operation
  }
}
```

Mapping operations: Example

- Rule for transforming leaf classes
  - selects only classes without subclasses, collects all the inherited properties and associations, creates new class in the target model

target type: instance created on call

```
mapping Class::copyLeafClass() : Class
  when {
    not hierarchical.allInstances(Generalization)->exists(g | g.general = self)
  } {
  name := self.name;
  ownedAttribute += self.ownedAttribute.
                    map copyOwnedProperty();
  ownedAttribute += (self.allFeatures()[Property] –
                     self.ownedAttribute).copyProperty(self);
  self.allFeatures()[Property]->select(p |
    not p.association.oclIsUndefined()).association.copyAssociation(self);
}
```

object on which mapping is called

call of another mapping

guard: mapping operation only executed for elements for which the guard expression evaluates to true

call of a helper

- Mappings only executed once
- Call of mappings with OCL-syntax (*collection*`->map` vs. *object*`.map`)

Helpers: Example

meta-model extension

```
intermediate property Property::mappedTo : Set(Tuple(c : Class, p : Property));

helper Property::copyProperty(in c : Class) : Property {
  log('[Property] name = ' + self.name);
  var copy := object Property {
    name := self.name;
    type := self.type.map transformType();
  };
  self.mappedTo += Tuple{ c = c, p = copy };
  return copy;
}
```

object creation and population

Resolving object references

- The transformation engine maintains links among source and target model elements. These links are used for resolving object references from source to target model elements and back.
  - `resolveIn` is an operation that looks for model elements of a given type (`Class`) in the target model derived from a source element by applying a given rule (`copyLeafClass`).

```
helper Association::copyAssociation(in c : Class) : Association {
  var theOwnedEnd : Property := self.ownedEnd->any(true); …
  return object Association {
    name := self.name;
    package := self.package.resolveoneIn(Package::transformPackage, Package);
    ownedEnd += new Property(theOwnedEnd.name,
                             c.resolveoneIn(Class::copyLeafClass, Class)); …
  }
}
```

call to constructor

  - Variants: `resolve(`i | exp`)`, `resolveone(`i | exp`)`
  - `late resolve` for resolving **after** the transformation (in order of calls)

Mapping operations: Disjunction, inheritance, merging

```
mapping DataType::copyDataType() : DataType {
  name := self.name;
  ownedAttribute += self.ownedAttribute.map copyOwnedProperty();
}

mapping PrimitiveType::copyPrimitiveType() : PrimitiveType {
  init {
    result := self.deepclone().oclAsType(PrimitiveType);
  }
}

mapping Type::transformType() : Type
  disjuncts DataType::copyDataType,
            Class::copyLeafClass,
            PrimitiveType::copyPrimitiveType;
```

- Inherited rules executed after `init`
- Merged rules executed after `end`

Imperative OCL constructs

- More sophisticated control flow
  - `compute (`*v* `:` *T* `:=` *exp*`)` *body*
    - like `let … in`
  - `while (`*cond*`)` *body*
  - *coll*`->forEach (`*i* `|` *exp*`)` *body*
  - `break`, `continue`
  - `switch`-statement, exceptions

MOFM2T

MOFM2T: OMG's model-to-text transformation standard

- **M2T** stands for **M**odel-**to**-**T**ext
  - OMG standard language for *transforming* MOF models into text

- Current status (June, 2011): version 1.0, formal/08-01-16

http://www.omg.org/spec/MOFM2T/1.0/

M2T Transformations: Example (1)

```
[comment encoding = UTF-8 /]
[** Java Beans-style code from UML static structure */]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')]

[**
 * Generate a Java file from a UML class
 * @param aClass
 */]
[template public generateClass(aClass : Class)]
[comment @main/]
[file (aClass.name.concat('.java'), false, 'UTF-8')]
public class [aClass.name/] {
[for (p : Property | aClass.attribute) separator('\n')]
[generateClassAttribute(p)/]
[/for]
}
[/file]
[/template]
```

metamodel type

top-level rule (several possible)

output in file, not appending

call of another template

verbatim text

M2T Transformations: Example (2)

```
[template public generateClassAttribute(aProperty : Property)]
private [getTypeName(aProperty.type)/] [aProperty.name/];

public [getTypeName(aProperty.type)/] [aProperty.name.toUpperFirst()/]() {
    // [protected(aProperty.name)]
    // TODO implement
    // [/protected]
    return this.[aProperty.name/];
  }
[/template]

[template public generateDataType(aDataType : DataType)]
[comment @main/]
[file (aDataType.name.concat('.java'), false, 'UTF-8')]
public class [aDataType.name/] [for (p : Property | aDataType.attribute)
                                before(' {\n') separator('\n') after('\n}')]
  public [getTypeName(aProperty.type)/] [aProperty.name/]; [/for]
[/file]
[/template]

[query public getTypeName(aType : Type) : String = aType.name /]
```

protected code message

output in file, not appending

top-level rule (several possible)

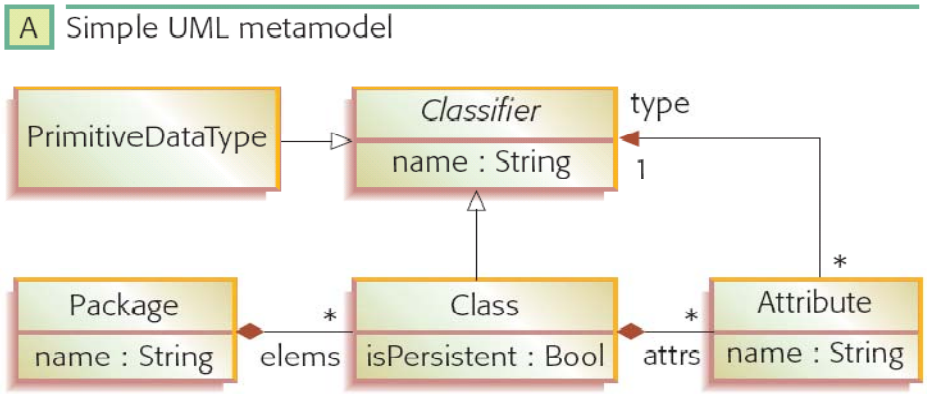
before first, in-between each, and after last item

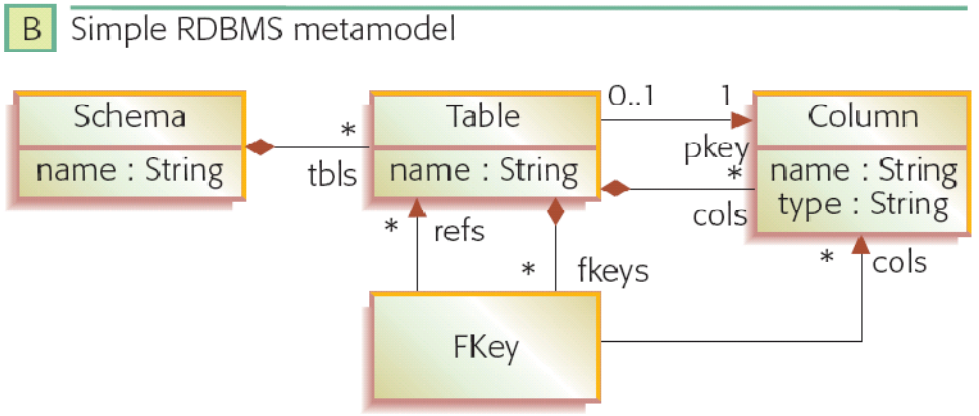
MOFM2T features

- Tracing
  - `[trace(`*id*`)]` … `[/trace]`
- Change of escape direction
  - `@text-explicit` (default, shown above)
  - `@code-explicit`
- Macros
- Module structure
  - public module elements visible outside a module
  - guards on templates for selecting a template when overriding (overridden template callable with `[super/]`)
- No type checking of output

Model Transformation Languages

Model-to-model approaches: Example

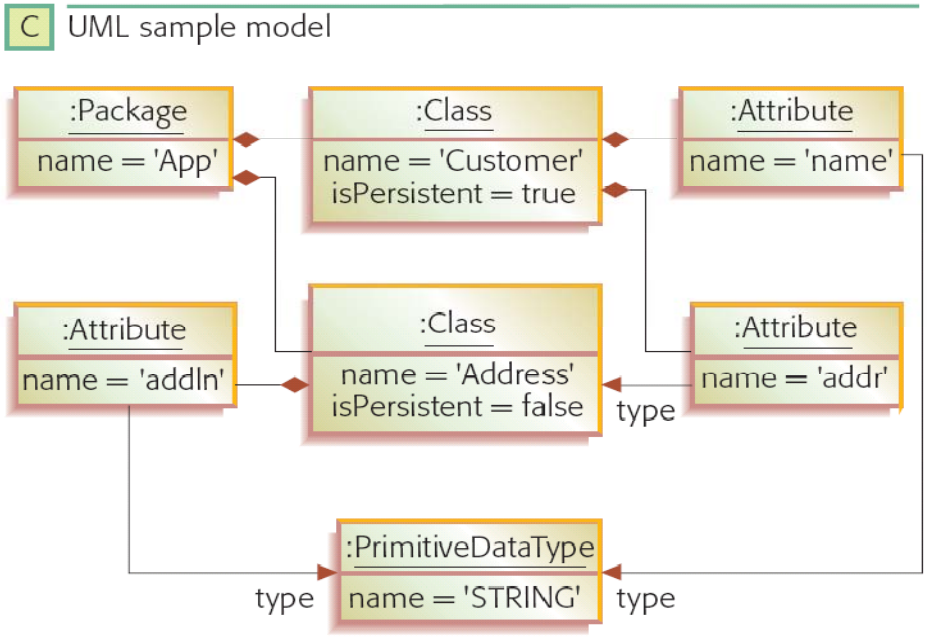


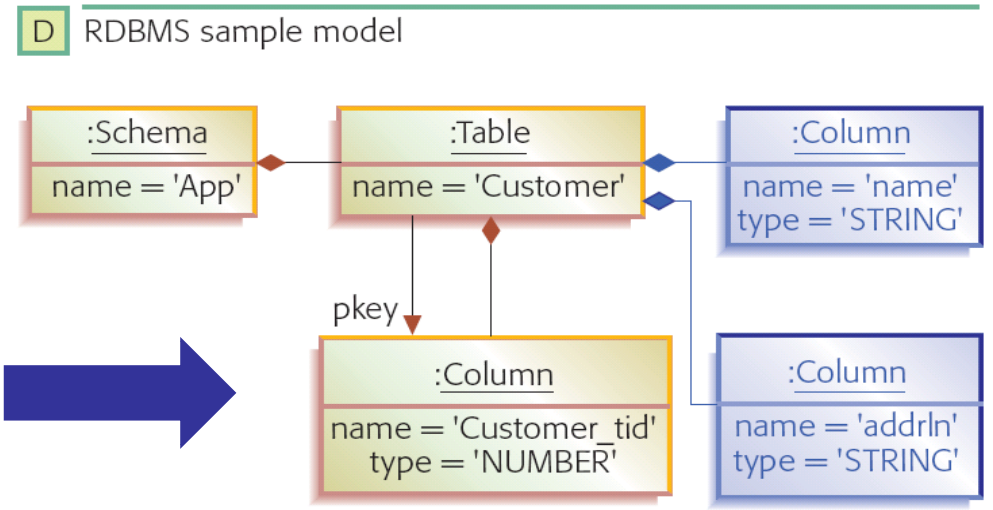


1. Package-to-schema
   - Every package in the class model should be mapped to a schema with the same name as the package.
2. Class-to-table
   - Every persistent class should be mapped to a table with the same name as the class. Furthermore, the table should have a primary-key column with the type NUMBER and the name being the class name with _tid appended.
3. Attribute-to-column
   - The class attributes have to be appropriately mapped to columns, and some columns may need to be related to other tables by foreign key definitions.

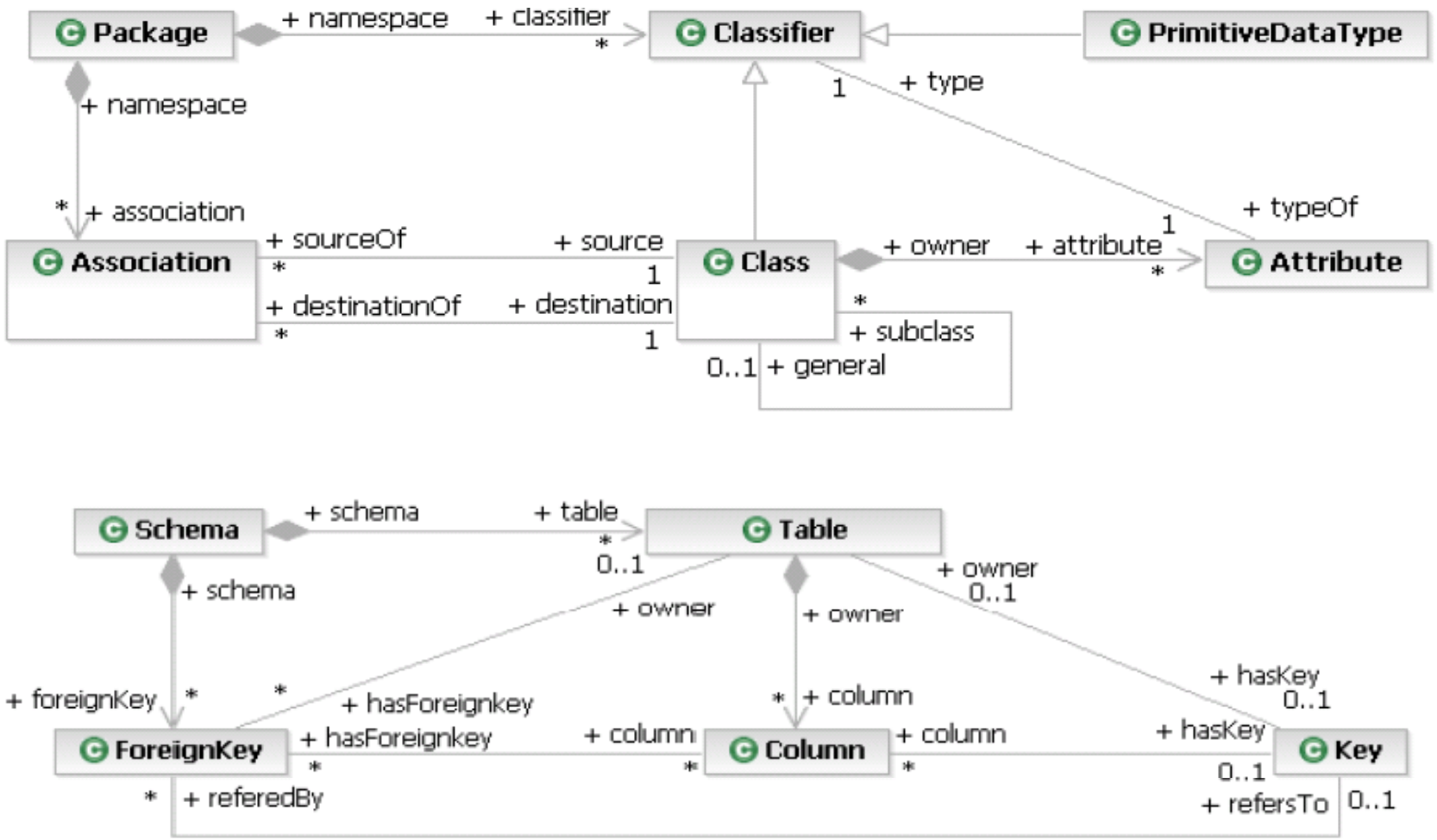
Model-to-model approaches: Example

1. Package-to-schema
2. Class-to-table
3. Attribute-to-column





UML to RDBMS example: Metamodel



ATLAS Transformation Language (ATL)

- **Hybrid** approach
  - declarative rules and imperative blocks
  - based on OCL
- Developed by ATLAS Group (INRIA & LINA)
- Integrated into Eclipse platform

http://www.eclipse.org/m2m/atl/

- **Modules** composed of
  - Rules
    - matched rules (top-level)
    - called rules
  - Helpers
- **Normal execution mode**: target model generated by explicit rules
- **Refinement execution mode**: target model generated by explicit rules + all model elements that are not changed by rules

ATL: Matched rules

- Pattern-based generation of target elements from source elements

```
rule rule_name {
  from in_var : in_type [(condition)]?
  [using {
    var1 : var_type1 = init_exp1;
    ...
    varn : var_typen = init_expn;
  }]?
  to
    out_var1 : out_type1 (bindings1),
    out_var2 : distinct out_type2 foreach(e in collection) (bindings2),
    ...
    out_varn : out_typen (bindingsn)
  [do {
    statements
  }]?
```

source pattern

local variables

target patterns

iterated target pattern

imperative block for changing target elements

ATL: Example (1)

```
module SimpleClass2SimpleRDBMS;
create OUT : SimpleRDBMS from IN : SimpleClass;

rule PersistentClass2Table {
  from c : SimpleClass!Class
             (c.is_persistent and c.parent->oclIsUndefined())
  using {
    primary_attributes :
      Sequence(TupleType(name : String,
                         type : SimpleClass!Classifier,
                         isPrimary : Boolean)) =
        c.flattenedFeatures->select(f | f.isPrimary);
    persistent_features : Sequence(TupleType(…)) = …;
    foreign_key_attributes : Sequence(TupleType(…) = …;
    rest_of_attributes :
      Sequence(TupleType(name : String,
                         type : SimpleClass!Classifier)) =
        c.flattenedFeatures->
          select(tuple | not tuple.isPrimary and
                         not tuple.type->oclIsKindOf(SimpleClass!Class));
  }
```

ATL: Example (2)

```
  to t : SimpleRDBMS!Table
           (name<-c.name,
            cols<-primary_key_columns->union(foreign_key_columns)->union(rest),
            pkey<-primary_key_columns,
            fkeys<-foreign_keys),
    primary_key_columns : distinct SimpleRDBMS!Column
                          foreach (primAttr in primary_attributes)
                            (name<-primAttr.name,
                             type<-primAttr.type.name),
    …
}

helper context SimpleClass!Class def :
  allAttributes : Sequence(SimpleClass!Attribute) =
    self.attrs->
      union(if not self.parent.oclIsUndefined()
            then self.parent.allAttributes->select(attr |
                   not self.attrs->exists(at | at.name = attr.name))
            else
              Sequence {}
            endif)->flatten();

…
```

QVT Relations: Language Overview

- Declarative language based on relations defined on model elements in meta-models
- Object patterns that may be matched and instantiated
- Automatic handling of traceability links
- Transformations are potentially multidirectional
- Supported execution scenarios:
  - Check-only: verifies if given models are related in a certain way
  - Unidirectional transformations
  - Multidirectional transformations
  - Incremental update of existing models
- Relations uses implicit rule scheduling which is based on the dependencies among the relations.
- The Relations semantics is divided into two steps
  - It first conducts a checking step, where it checks, whether there exists a valid match in the target model that satisfies the relationship with the source model
  - On the basis of the checking results, the enforcement semantics modifies the target model so that it satisfies the relationship to the source model

Relations transformations

- Relations **transformations** are specified between candidate models as a set of relations that must hold for the transformation to be successful. A **candidate model** is any model that conforms to a model type.
- In a **relation**, **domains** are declared that match elements in the candidate models.
  - Relations can be further constrained by two sets of **predicates**, a when clause and a where clause. The
    - The **when** clause specifies the conditions under which the relationship needs to hold
    - The **where** clause specifies the condition that must be satisfied by all model elements participating in the relation.
- Each of the domains is also associated with several **object template expressions** used to match **patterns** in the candidate models
  - **Pattern matching** is the process to determine correspondences between the candidate models
- **Checkonly** and **enforce** determine in which direction the transformation is executed.
- Existing objects are updated. For this purpose, the concept of **keys** uniquely identify object instances.

QVT Relations: Graphical syntax

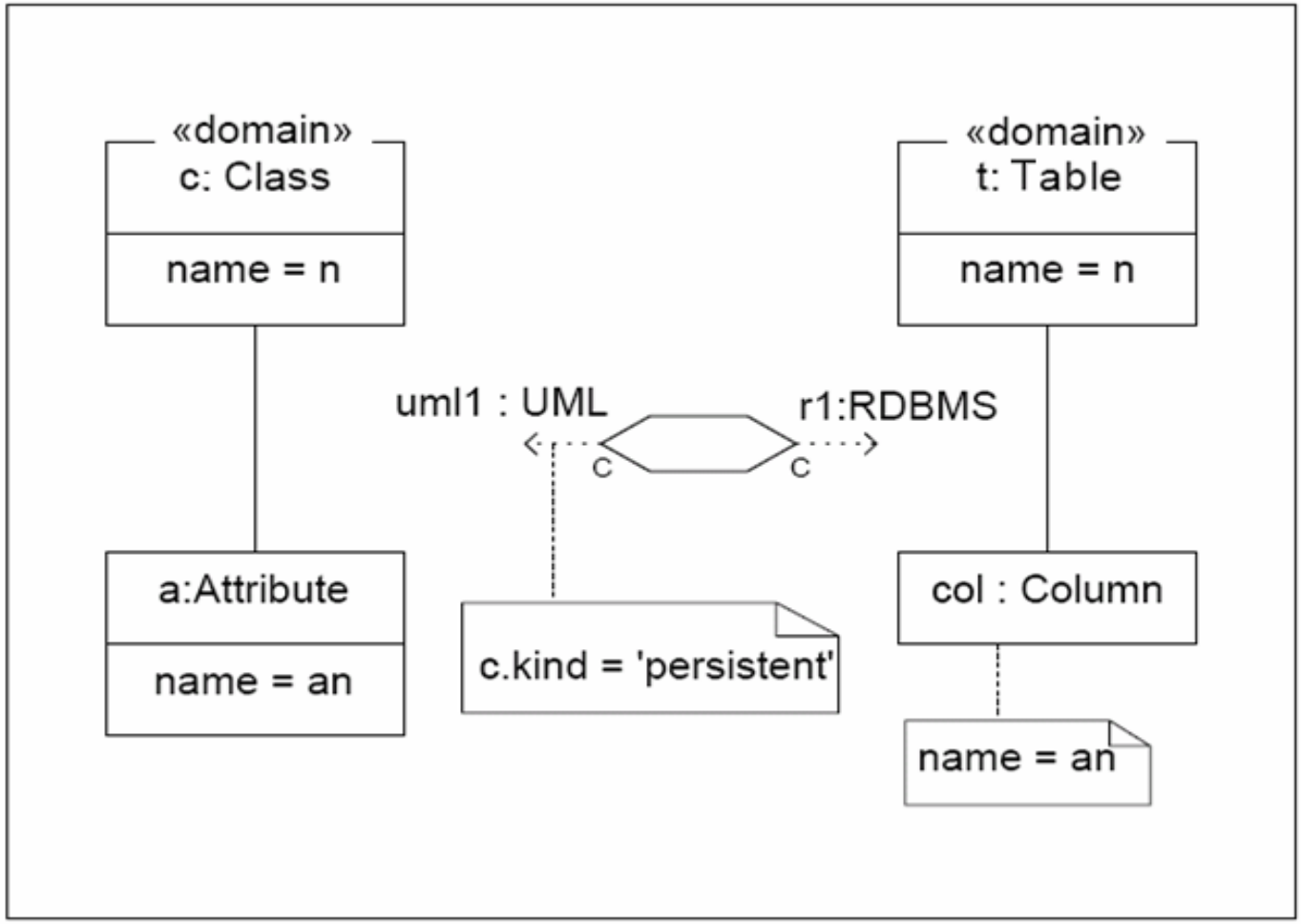


Figure from [QVTP]

Relational approach: QVT Relations (1)

```
top relation ClassToTable {
  cn : String; prefix : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlNamespace = p : SimpleUML::UmlPackage { },
    umlKind = 'Persistent', umlName = cn
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
    rdbmsName = cn,
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn + '_tid', rdbmsType = 'NUMBER'
    },
    rdbmsKey = k : SimpleRDBMS::RdbmsKey {
      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}
    }
  };
  when { PackageToSchema(p, s); }
  where { ClassToPkey(c, k); prefix = cn;
          AttributeToColumn(c, t, prefix); }
}
```

Relational approach: QVT Relations (2)

```
relation AttributeToColumn {
  checkonly domain uml c : SimpleUML::UmlClass {
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
  };
  primitive domain prefix : String;
  where {
    ComplexAttributeToColumn(c, t, prefix);
    PrimitiveAttributeToColumn(c, t, prefix);
    SuperAttributeToColumn(c, t, prefix);
  }
}
```

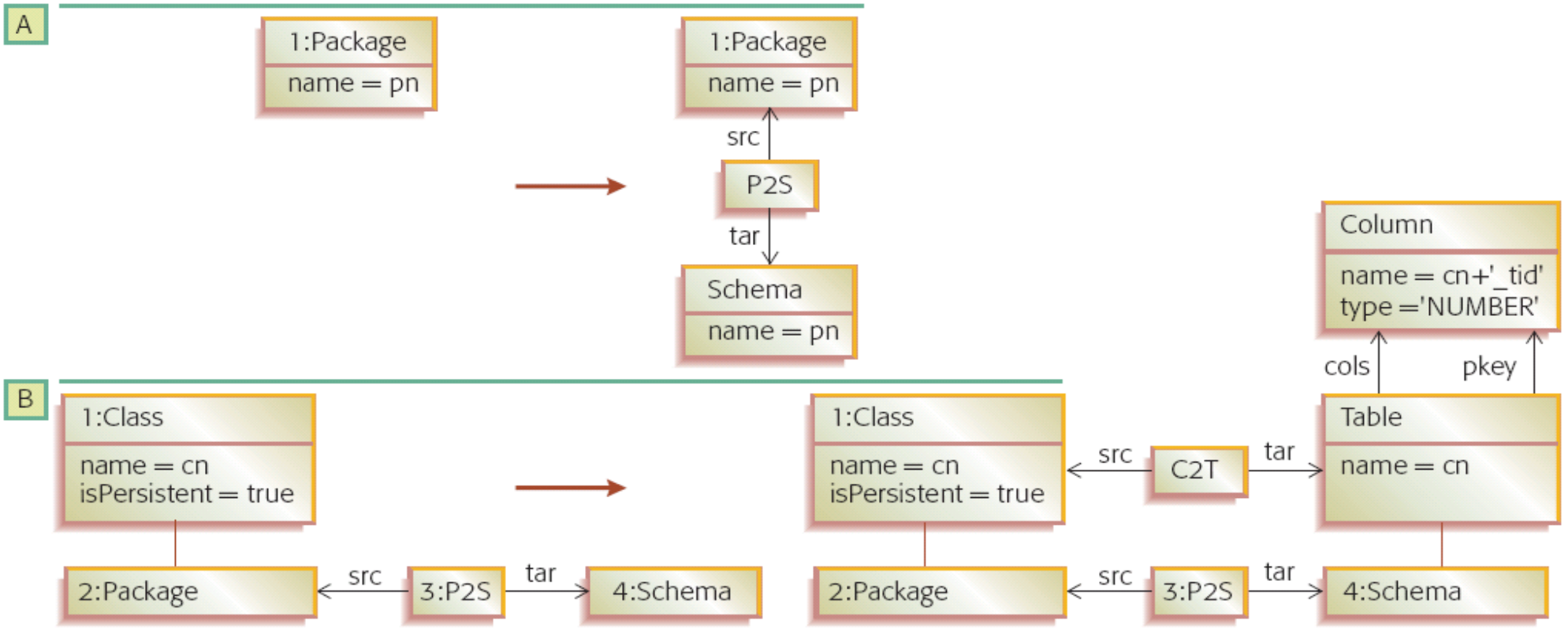
ternary relation

Relational approach: QVT Relations (3)

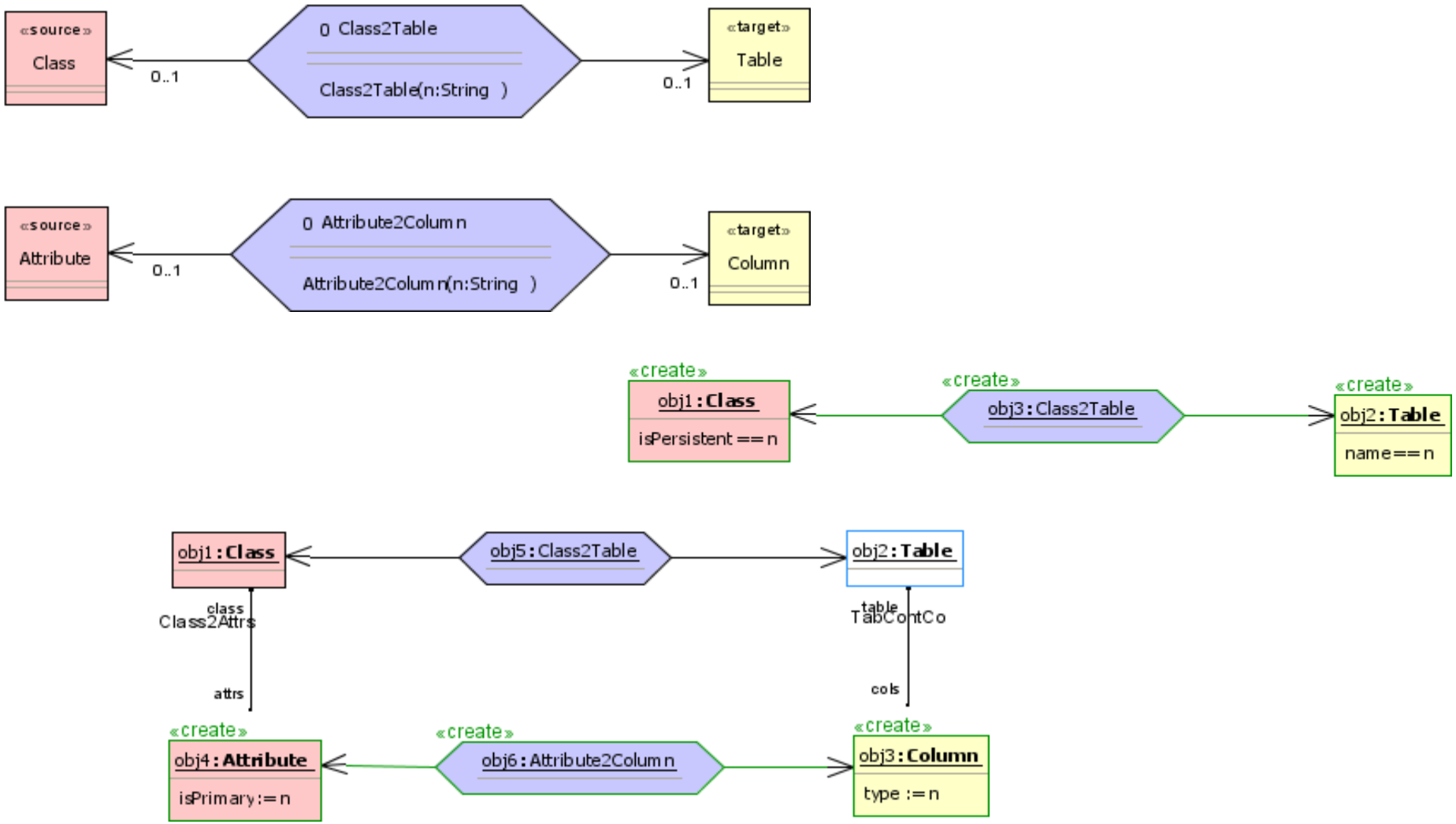
```
relation ComplexAttributeToColumn {
  an : String;
  newPrefix : String;

  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,
      umlType = tc : SimpleUML::UmlClass { }
    }
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
  };
  primitive domain prefix : String;
  where {
    newPrefix = prefix + '_' + an;
    AttributeToColumn(tc, t, newPrefix);
  }
}
```

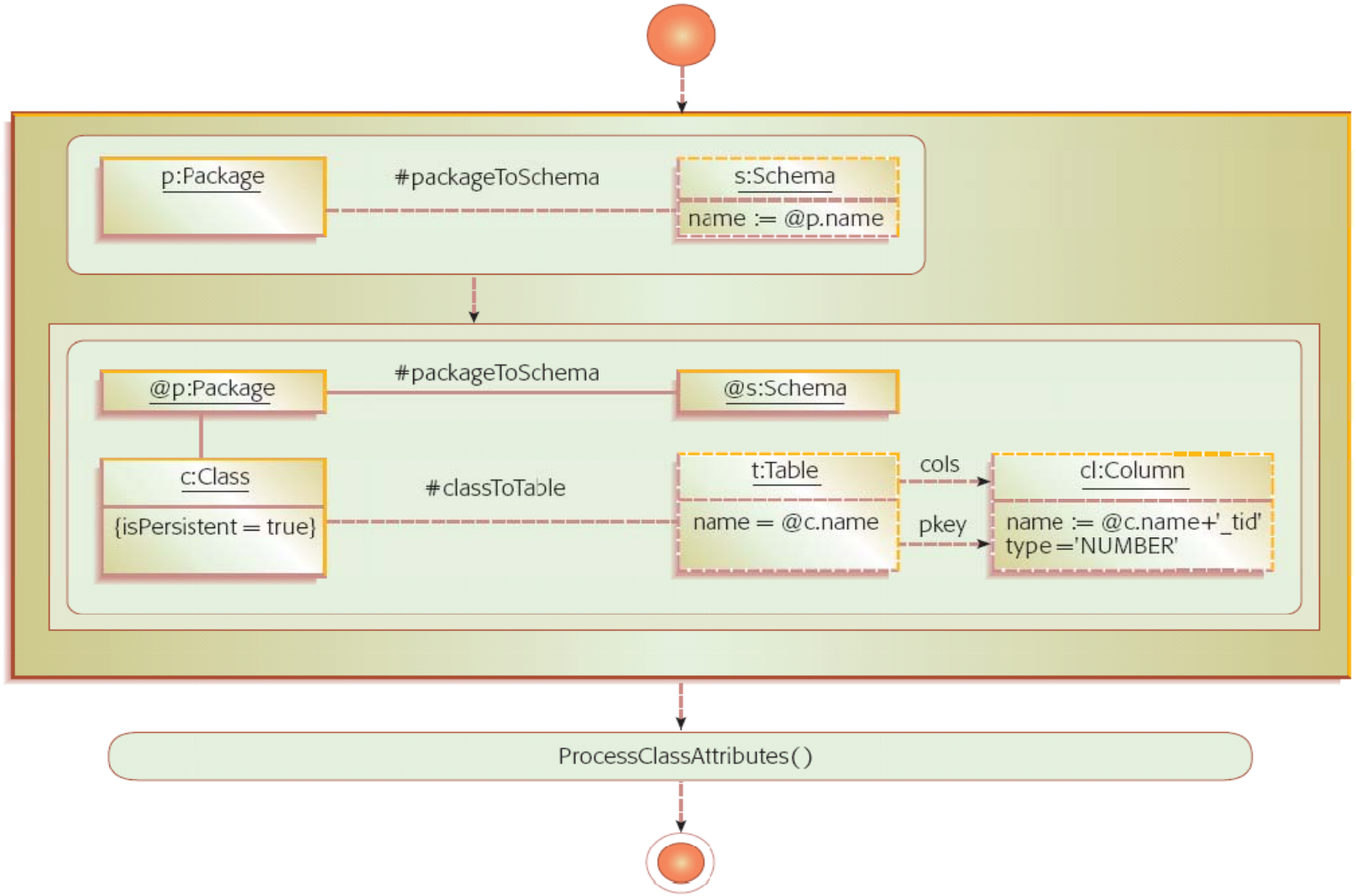
Graph-transformation approach: AGG



Graph-transformation approach: MOFLON



Graph-transformation approach: Mola



Model-to-model approaches: Comparison (1)

		ATL	QVT Rel.	QVT Op.	MOFLON	AGG
Transformation scenarios	Model synch-ronisation	×	✓	×	×	×
	Conformance checking	×	✓	×	×	×
	Model transformation	✓	✓	✓	✓	✓
	In-place update	✓	✓	✓	✓	✓
	Interactive transformation	×	×	×	×	✓
Paradigm	Declarative	✓	✓	×	✓	✓
	Hybrid	✓	×	×	×	×
	Imperative	✓	×	✓	×	×
Directionality	Unidirectional	✓	✓	✓	✓	✓
	Multidirectional	×	✓	×	✓	×

Model-to-model approaches: Comparison (2)

		ATL	QVT Rel.	QVT Op.	MOFLON	AGG
Cardinality	M-to-N	✓	✓	✓	✓	×
	1-to-1	✓	✓	✓	✓	✓
Traceability	Automatic	✓	✓	✓	✓	×
	User-specified	×	×	×	✓	✓
Query language		OCL-based	Object patterns	OCL-based	Graph patterns	Graph patterns
Rule scheduling		implicit, explicit	implicit	explicit	implicit	implicit, explicit
Rule organisation		inherit., libraries	inherit.	inherit.	layering	layering
Reflection	runtime access to transf.	×	×	×	×	×

Java Emitter Templates (JET)

- Template-based model-to-text transformation approach
  - avoiding to write repetitive glue code
  - code generation from Java objects
  - transformation of XML, XMI
    - integrated with EMF
- Like Java Server Pages (JSPs)
  - expressions (<%= ... %>)
  - scriptlets for inserting arbitrary Java statements (<% ... %>)
  - JET translated into Java class behind the scenes
- JET1
  - `generate(Object argument)`
- JET2
  - `generate(JET2Context context, JET2Writer out)`

JET2: Example — Template

```
<%@jet package="purchase"
       class="PurchaseOrderTest"
       imports="java.util.*" %>

<% PurchaseOrder order = (PurchaseOrder)context.getSource(); %>

<HTML>
<HEAD>Purchases</HEAD>
<BODY>
<P>Order to: <%=order.getShipTo()%> (bill to: <%=order.getBillTo()%>)
<UL>
<% for (Item item : order.getItems()) { %>
<LI>Item <%=item.getProductName()%>
<% }%>
</UL>
</P>
</BODY>
</HTML>
```

name of generated class

access to input

scriptlet

expression

JET2: Example — Generated code

```
public void generate(final JET2Context context, final JET2Writer __out) {
  JET2Writer out = __out;
  out.write("<?xml version=\"1.0\" encoding=\"utf-8\"?>");  //$NON-NLS-1$
  out.write(NL); out.write(NL);
  PurchaseOrder order = (PurchaseOrder)context.getSource();
  out.write(NL); out.write("<HTML>");  //$NON-NLS-1$
  out.write(NL); out.write("<HEAD>Purchases</HEAD>");  //$NON-NLS-1$
  out.write(NL); out.write(NL); out.write("<BODY>");  //$NON-NLS-1$
  out.write(NL); out.write("<P>Order to: ");  //$NON-NLS-1$
  out.write(order.getShipTo()); out.write(" (bill to: ");  //$NON-NLS-1$
  out.write(order.getBillTo()); out.write(")");  //$NON-NLS-1$
  out.write(NL); out.write("<UL>");  //$NON-NLS-1$
  out.write(NL);
  for (Item item : order.getItems()) {
    out.write("<LI>Item ");  //$NON-NLS-1$
    out.write(item.getProductName());
    out.write(NL);
  }
  out.write("</UL>");  //$NON-NLS-1$
  out.write(NL); out.write("</P>");  //$NON-NLS-1$
  out.write(NL); out.write("</BODY>");  //$NON-NLS-1$
  out.write(NL); out.write("</HTML> ");  //$NON-NLS-1$
}
```

JET2: Example — Transformation

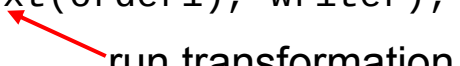
```
PurchaseFactory purchaseFactory = PurchaseFactory.eINSTANCE;

PurchaseOrder order1 = purchaseFactory.createPurchaseOrder();
order1.setBillTo("A");
order1.setShipTo("B");

Item item1 = purchaseFactory.createItem();
item1.setProductName("X"); item1.setPrice(100.0f); item1.setQuantity(3);
item1.setOrder(order1);

Item item2 = purchaseFactory.createItem();
item2.setProductName("Y"); item2.setPrice(200.0f); item2.setQuantity(2);
item2.setOrder(order1);

JET2Writer writer = new BodyContentWriter();
new PurchaseOrderTest().generate(new JET2Context(order1), writer);
System.out.println(writer.toString());
```



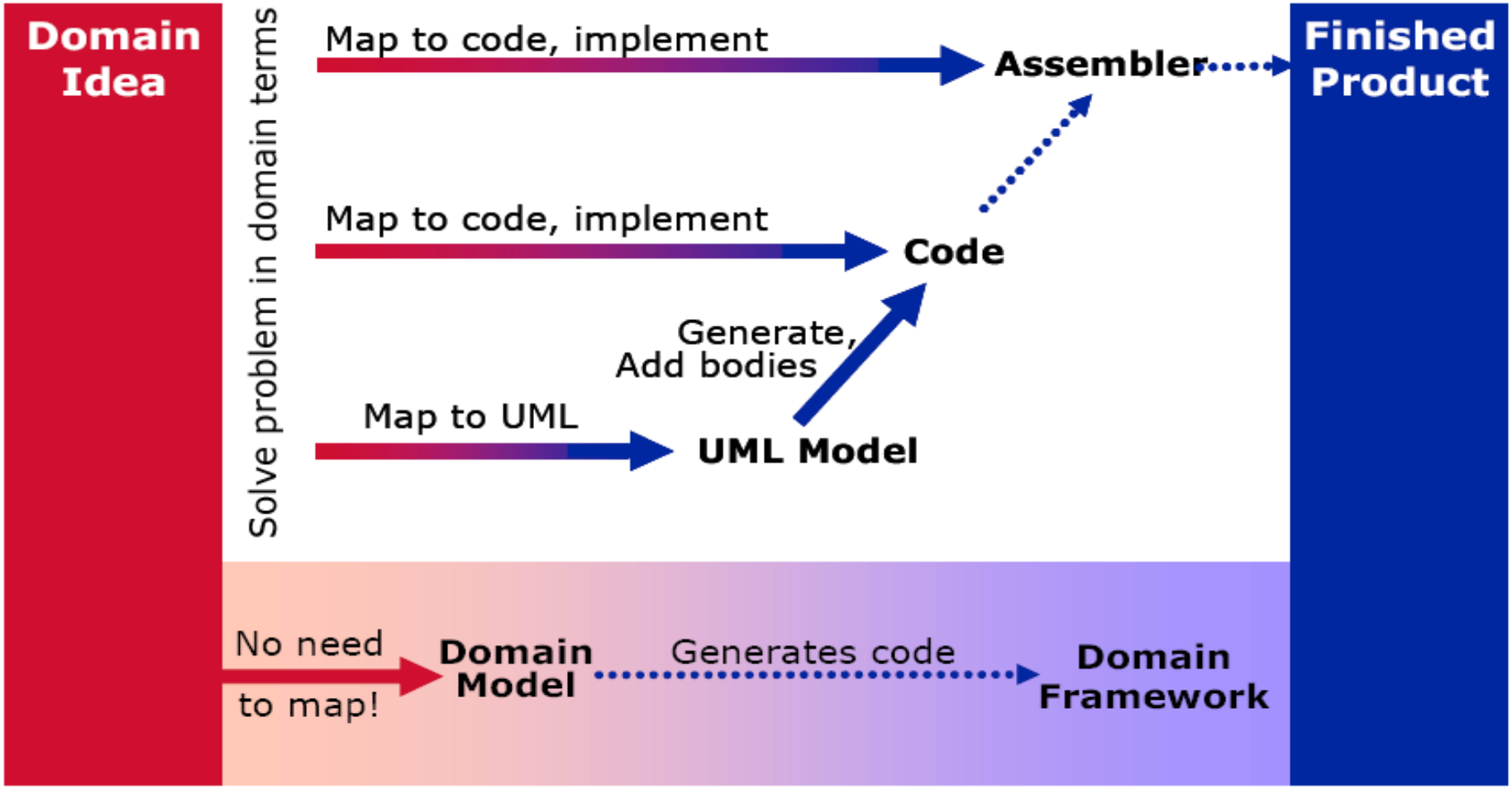
Domain-Specific Languages

UML – one size fits all?

- While the OMG MDA promotes UML as the visual “universal” glue suitable for modelling everything, there exists also a trend towards development and co-existence of several **domain-specific modelling languages (DSLs)**.

- UML is seen as a “general-purpose” language while DSLs may be more expressive for most purposes.
- A model-driven framework needs to acknowledge the existence of different models and views expressed in different modelling languages.
- The MDA technologies (MOF, UML) can help to align these models through a common (meta-)meta-modelling language (MOF) on which model transformations and model mappings can be defined.

Domain-Specific Languages



Advantages of using UML profiles

- UML is open standard language: many available books and training courses.
- UML is a recognized and transferable skill for software developers
- UML profiles provide a lightweight approach that is easily implemented using readily available UML tooling.
- Models with UML profiles applied can be read by all UML tools, even if they don't have any knowledge of the profile.
- Basing all DSLs on UML creates a set of related languages that share common concepts.
  - makes new profiles more readily understandable
  - enables models expressed by different DSLs to be integrated easily

Disadvantages of using UML profiles

- New meta-models are adjusted to specific user groups, application domains, and usage context
  - UML profiles only permit a limited amount of customisation
  - New modelling concepts that can be only expressed by extending existing UML elements
- In DSLs the semantics of the modelling language is better understandable to the users of the application domain. The scope of DSLs is customized to its application domain and use
  - User will be guided by the modelling language towards certain types of solutions
  - The use of UML does require familiarity with modelling concepts.
- It is necessary to restrict the usage of UML with UML profiles, since most of UML usages only rely on a small subset of the entire meta-model
  - In general is much more difficult to work by restriction than by extension (developing new meta-models
  - Working by extension fosters the automation of code generation, since code generation does have to take into account less modelling and interpretation possibilities

Rationale for Using Profiles vs. MOF (benefits)

- Profiles
  - are used for extending the UML language (the “reference meta-model”)
  - are supported by UML Case tools
  - guarantee the UML conformance of the extensions
  - provide a dynamic extension capacity (i.e. extending an existing model)
  - Typical example: *UML for a certain purpose*

- MOF extensions
  - are used to create new meta-models
  - apply to any meta-model
  - New models are created from MOF extensions (no existing model updates)
  - are supported by meta-CASE tools or infrastructure
  - Typical example: *New meta-model* (e.g. DSLs for workflows, services etc.)

Meta-model characteristics

- Suited for target roles
  - Support domain concepts and scenarios of target roles
  - Ease-of-use and understandable for modeler (use terms)
  - Support precise details and correctness for solution architect
- Avoid unnecessary complexity
  - Keep it simple, stupid (KISS)
  - Number of elements and associations
  - Type and navigation of associations
- Make it modular
  - Provide core with extensions
  - Define and illustrate possible subsets (”dialects”) that support scenarios
  - Consider integration and extension points
- Suited for implementation
  - EMF representation
  - Transformation from/to UML profile
  - Transformation to PSM/Code